The Case for a Functional Internet of Things

Till Hänisch
DHBW Heidenheim
Marienstrasse 20
89518 Heidenheim
++49 7321 2722-292
haenisch@dhbw-heidenheim.de

ABSTRACT

In this paper a case study is presented which shows that the code size and complexity of a system which collects and interprets sensor data in an Internet of Things scenario can be reduced using functional programming techniques. On the one hand this is especially important for security reasons: Such a system must run for a long time without an effective way to distribute software patches. On the other hand in this kind of system the consequences of a malfunction (intended or not) are much more critical than in standard computing situations, because real world buildings or industrial sites are affected.

From a high level perspective the data processing at the base station of such a sensor network can be considered as a set of mathematical functions operating on a stream of values. Each function creates a new stream of values, which might be processed by another function. This means that the complete functionality can easily be described and programmed in a functional language, such as elixir, Erlang or Scala.

Keywords
Internet of Things, Functional programming, Software Architecture, IT-Security

1. INTRODUCTION

Internet of Things appliances, such like light switches, thermostats or other kinds of sensors or actors, are especially sensitive to software errors. While minor malfunctions may be acceptable, software bugs might lead to security problems, which are not acceptable, since they will have consequences in the real world.

Today's method of keeping systems, e.g. operating systems, secure is to patch them permanently to solve security problems. That is not practical for the Internet of Things (IoT). The neccessity to patch on a regular base combined with the long lifespan of components like building automation systems would result in a severe configuration management problem: It is almost impossible

to properly test systems composed of that many components, with different hardware and software versions. Constant updates will sooner or later result in interoperability problems. Even automatic patching will not solve this issue.

Since regular updates are not feasible, a different way of keeping the system secure is required. There are basically two ways to achieve this. One possible solution is a self healing system. While there are different research efforts to develop methods for creating such systems, there are no practical solutions yet. The Defense Advanced Research Projects Agency (DARPA) has identified this option of addressing security problems and started the Cyber Grand Challenge in 2013 to stimulate research about self healing networks (DARPA, 2013). In 2015 DARPA launched an initiative called Building Resource Adaptive Software Systems (BRASS) to build "software systems and data to remain robust and functional in excess of 100 years" (DARPA, 2015). The only way to achieve this is enabling self adaptive systems which adapt themselves to changing environments. While this might lead to interesting results in the future, this solution is not available for current systems.

Without usable techniques to automatically solve security problems, it is desirable to keep the number of bugs close to zero. One way to lower the number of bugs is small code size and low complexity. Fewer lines of code and lower coupling, especially as few side effects as possible, means fewer bugs. The question is, how to achieve that.

2. ARCHITECTURE OF INTERNET OF THINGS APPLICATIONS

In the simplest case, and only this case will be considered here, IoT means, that things talk to the internet. There are two common architectures for this kind of system: The first and simplest is a sensor node that is directly connected to the internet, typically by WLAN (Figure 1a). This requires a WLAN interface and in most cases an operating system that provides the necessary functionality. Typical hardware platforms for this kind of applications are either open, complex platforms like Raspberry Pi, Intel Galileo or Carambola, running some kind of Unix or Windows OS. These systems are flexible and powerful, however they require a continuous power supply since their energy consumption of up to 15 Watt (Reese, 2015) cannot be delivered by batteries.

Figure 1. *a) Sensor node transmitting directly to the Internet, b) Sensor node transmitting to a base, which transmits data to the internet*

a) b)

In the second kind of applications, small battery powered sensors like fitness trackers or sensor nodes send their data to a base station (Figure 1b).

The base station can either be a smartphone, a PC or a special appliance depending on the technical requirements of the system. Data is collected here and can be made accessible via the Internet or at least locally via standard internet protocols.

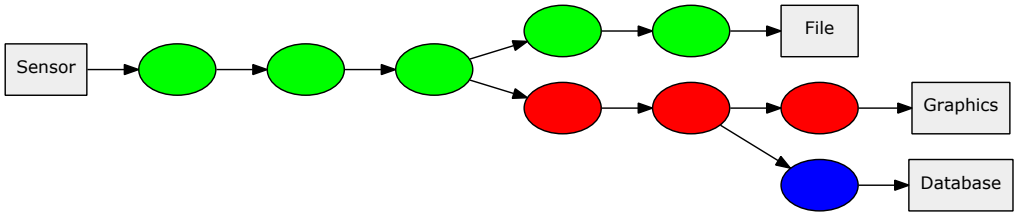
The disadvantage of the second architecture is the base station required in addition to the sensor nodes. The advantage is, that the sensor nodes can be very simple, and might not even require an operating system. This means, they can be cheap and battery powered. This is the scenario discussed in this paper.

3. FUNCTIONAL PROGRAMMING FOR THE IOT

In most cases the structure of sensor data is rather simple: Typically a measurement consists of a value, a timestamp, an identification of a location and a sensor id (what, when, where, who). This data can then be stored in a database or a flat file since usually no transactions are needed. Data is written only once and only sequentially.

Data is read by different applications at different times for different purposes. However, most data will be processed only once, at the time of creation. Data is usually analyzed in some way at this time, for example to detect anomalies, to generate statistical distribution parameters, to aggregate the data for later time series analysis and to display a graphical representation of the current data with a surrounding context.

Figure 2. *Sensor values considered as streams. Functions generate additional streams (different colors) from the original sensor stream.*



All of these processing activities can be seen as the application of a mathematical function on a set of sensor values, with the special case that the set contains only one value. Thus, the functionality of a processing node for an IoT application can be considered as a set of mathematical functions operating on a stream of values (Newton, 2004). Each function creates a new stream of values, which might be processed by another function. Some of the value streams will be archived in a datastore. That means, that the complete functionality can easily be described and programmed in a functional language like elixir, Erlang, Scala or Haskell.

There is a considerable debate about the advantages of using functional programming languages or at least functional programming techniques. Many languages adopt functional features to allow using functional techniques in the preferred environment, for example (Subramaniam, 2014). This debate is not new (Gat, 2000). In Gat's classic experiment it was shown, that many properties of programs like programmer productivity, performance etc. were better when the programs were written in Lisp, a very old functional language, compared to Java, a then modern imperative language.

Functional programming languages (and their environments like Erlang/OTP) are very good for writing reliable, highly concurrent applications with many concurrent processes and especially process failures (Armstrong, 2010). Writing applications like that was the reason for the development of the Erlang ecosystem in telecommunication systems like phone exchanges.

The same reasons for using functional languages in these environments are given in IoT scenarios. Concurrent event sources, e.g. sensor modules, unreliable communication with spurious errors because of wireless data transmission and a system that has to work highly reliable under any of these problems. Even if some sensors in a building or a factory setting are not working correctly, the data and data transformation must continue at least with the undisturbed data, the main control flow must not be affected by errors in other parts of the system. Nobody would tolerate a building where you can not turn on the lights, because a thermostat node crashes.

But this is not the most important point for choosing functional languages. A much stronger advantage of functional languages is, that the code for transformations like the ones described above, is much more concise than with traditional imperative languages. Although there is no formal proof for this assumption, there is a large number of anecdotal cases, for example from (Ford, 2013) or the case study described in a later section of this paper. An impressive case is John Carmack from ID software, who reimplemented Wolfenstein 3D in Haskell and found, besides other promising benefits, that the code size was reduced significantly (Carmack, 2013).

Short code without side effects (pure functional languages don't have side effects) is easier to verify for correctness than imperative code. That means, it contains fewer errors. While there is a significant, but only small correlation

between the programming language and the error rate, there is a clear dependency between code size and error rate (Ray, 2014). Since programs written in functional languages tend to be shorter than programs written in imperative languages, they should contain fewer errors.

Fewer errors means less security problems, which is the main point. Internet of Things applications have a direct relation to the real world. Security problems in this context mean not only damage files on a disk, which might be restored from a backup, but cause damage and or monetary loss in the real world.

If someone hacks your thermostat while you are on vacation and sets the temperature to maximum all the time, your heating system will go full speed for weeks. That means a substantial financial risk. If someone hacks your home security system and locks the front door, so you can not enter your house at night, that would be very unpleasant. If someone hacks your car and turns the headlights off while you are driving at 100 km/h, that is a substantial security risk and so on.

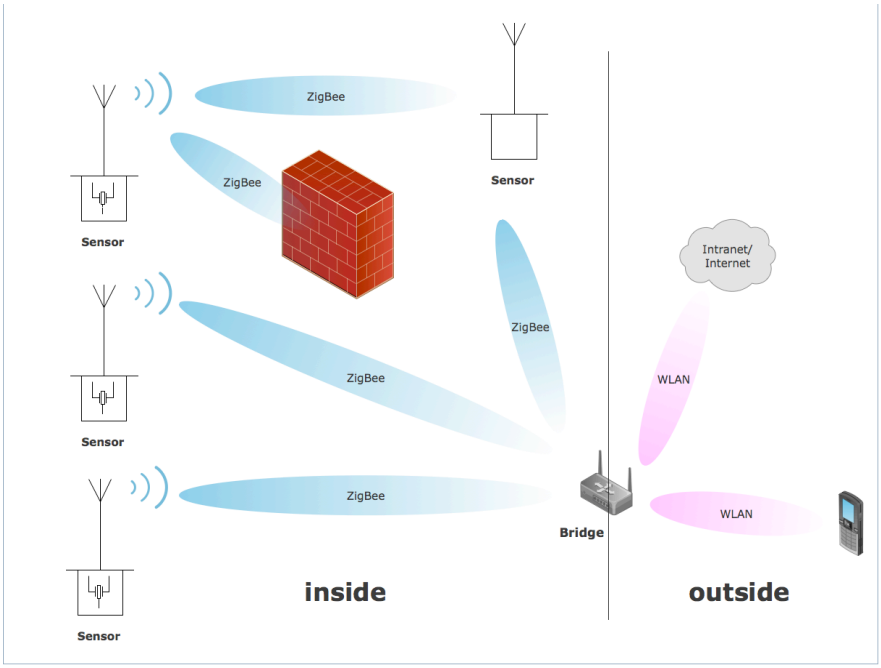
To limit the possible damage by security problems in IoT applications, it is either neccessary to develop and deploy a widely accepted platform, that has few bugs and is constantly updated throughout the world like for example Apple does with iOS or we need as much diversity in these systems as we can get to reduce the risk of a complete failure (Schneier, 2010). That means individually developed software with as few bugs as possible. And that means short, simple programs, which are easy to test and verify.

In the following chapter a case study is presented which shows that the code size and complexity for a systems which collects and interprets sensor data in an IoT scenario can be reduced using functional programming techniques.

4. CASE STUDY

The case discussed in this paper is a low cost low power sensor network to save energy in paper machines. By using wireless sensors for measuring temperature and humidity in the dryer section of a paper machine it is possible to optimize energy consumption by adjusting heating and air flow. Because the sensors need to be battery powered and send the data almost in real time for monitoring purposes, a low power network technology is needed, in this case ZigBee. The data is sent to a base station in packets with no guaranteed delivery, resulting in an at most once semantic. This results in some complexity of the base station code, which consists mainly of error handling and monitoring or logging functions.

Figure 3. *Architecture of the sensor network consisting of the sensor nodes and a base station ("bridge")*



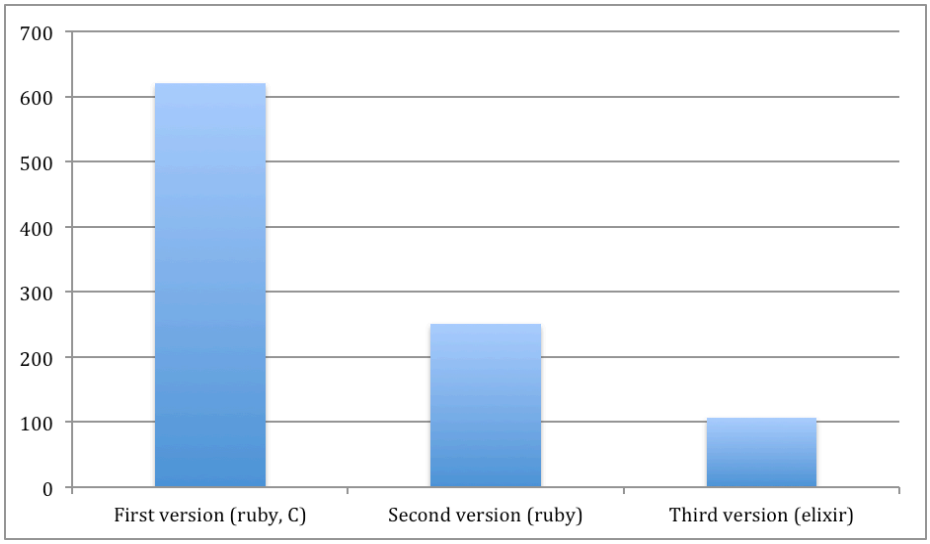
In this article, only the code running on the base station (see Figure 3) will be considerer, the code on the sensor nodes mainly handles communication with the sensor hardware and has a very simple structure since no data is stored locally. In more complex cases this part could also be implemented using functional programming techniques like functional reactive programming (Khare, 2015).

The original version consists of 620 lines of source code (262 lines of C, 355 lines of ruby, 3 lines of python) plus a few shell scripts for startup tasks etc.

By reevaluating the user requirements the code size could be reduced to 251 lines of ruby (including comments and empty lines, that is 225 nonempty lines resp. 194 nonblank lines without comments). That is some 40% of the original size. The reimplementation in elixir resulted in 106 lines of code (including comments and empty lines, that is 86 non-blank lines resp. 68 lines of code without comments). That is some 42% of the second version, 17% of the original version.

Remark: The Erlang version was about the same size as the elixir version (which is no surprise, since it has the same structure, the same functions etc.) but felt somewhat alien at least to the author and was dropped in favour of the elixir version.

Figure 4. *Code size in lines of the different versions*



That means, that the code size was reduced by a factor of 5.

Figure 5. *Listing of the most complex function of the elixir version*

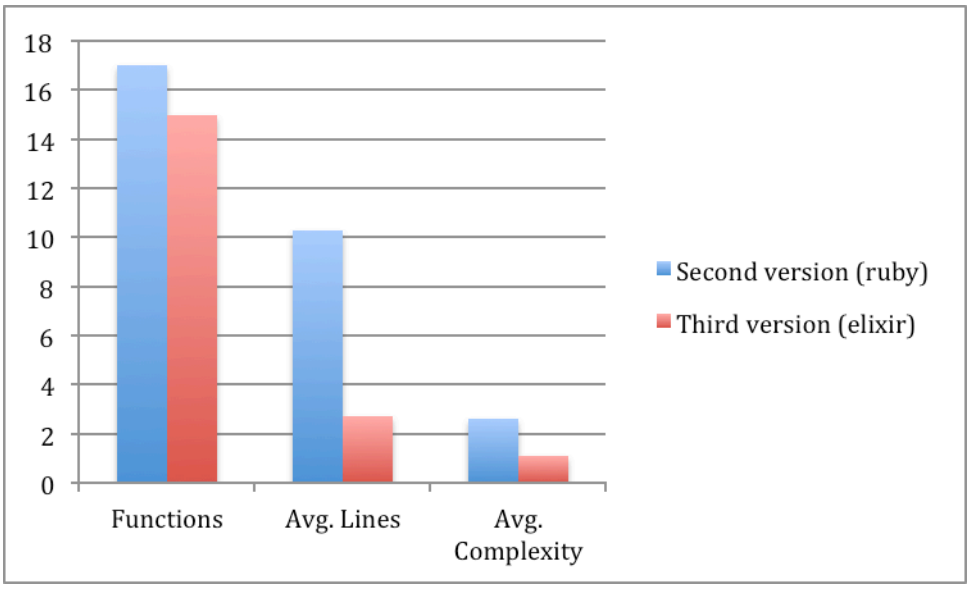
```
def process_file(input_file,current_values) do
  task = Task.async(fn -> IO.read(input_file, :line) end)
  try do
    row = Task.await(task,5000)
    if (row != :eof) do
      new_values = process_line(
        String.split(String.rstrip(row), " "),current_values)
      process_file(input_file,new_values)
    end
  catch
    :exit, _ -> IO.puts "timeout"
      process_file(input_file,current_values)
  end
end
```

The final elixir version consists of 15 functions with an average length of 2.7 lines (only 6 functions have a length of 1 line). Only one function has a cyclomatic complexity greater than 1, this is the function shown in figure 5. This function handles timeouts when receiving data, so a complexity greater than one is mandatory.

The ruby versions consists of 3 classes with 17 methods having an average length of 10.3 lines. The average cyclomatic complexity is 2.6, the maximum cyclomatic complexity is 10. These numbers show, that the elixir version is not only much smaller (42 % of the size of the ruby version) but also the complexity is lower by a similar factor. According to (Watson, 1996) cyclomatic complexity correlates with the number of errors in software

modules. So the elixir version should have less errors than the much longer and more complex ruby version.

Figure 6. *Complexity measures of the ruby and elixir versions with the same functionality*



The original version (ruby and C) was replaced by the second, simplified ruby version because it showed a number of critical errors which were hard to find because they occured only rarely. This lead to a simplified ruby version which worked over a period of nearly two years with only two non critical bugs. The elixir version is running for only a few months now and showed no bug til today.

5. DISCUSSION

The problem with a study with n=1 are well known, see for example (Harrison, 2000). But experiments in software engineering are hard to do: Controlled experiments with n>1 would give better results, if and only if both samples are from the same basic population. This basic population must be representative for the real word. This is the problem with the controlled experiment approach. Usually experiments are done with voluntary students, but it would be difficult to find students which have the same amount of experience level, in ruby and elixir in this case. Typically someone knowing those two languages has way more experience with ruby than with elixir, since elixir is newer. Programmers knowing elixir or Lisp, Scala, Erlang will tend to have a more theoretical background than the typical developer of embedded systems, but much less practical experience. So even an experiment with a large number of participants would be of limited use for real projects.

The size of the example described above is much smaller than typical industrial projects. So the only firm conclusion that may be drawn from this case is that further, larger experiments are needed. On the other hand the processing of data in Internet of Things scenarios might (and probably should) (Namiot, 2014) be implemented as microservices, with a size comparable to the case described here.

Both of these points are valid, but controlled experiments with realistic project sizes are very hard to do: The group of people who would volunteer to work for a few years on a software project that is developed by a large number of other teams concurrently just to get some statistically valid data about program complexity is limited and certainly not representative for real world software engineers. So this problem is unsolvable and we will have to stay with small n=1 case studies.

Using the cyclomatic complexity as a measure for the expected number of errors in code is debatable, see for example (Abran 2004). On the other hand it is widely accepted and used in tools to measure complexity for exactly this purpose. In conclusion the correlation might not be absolutely proven, but in real world experience it works and it is plausible: The more paths in the code, the harder to understand and test, the harder to understand and test, the more errors.

6. CONCLUSION

Using functional programming techniques and/or languages can reduce the code size and the complexity of Internet of Things applications. Reduced code size and complexity means less bugs, that means less security problems.

Functional programming techniques fit well to the architecture of Internet of Things applications. It is therefore plausible that the described reduction in code size and complexity could be realized in other projects as well.

Elixir seems to be a good choice as an implementation language for Internet of Things applications.

7. REFERENCES

Abran, A., Lopez, M., and Habra, N. 2004. *An Analysis of the McCabe Cyclomatic Complexit Number*, Proceedings of the 14th International Workshop on Software Measurement (IWSM) IWSM-Metrikon, 2004, Magdeburg, Germany: Springer-Verlag, pp. 391-405.

Armstrong, J., 2010. *Erlang*, Communications of the ACM, Volume 53 Issue 9, September 2010

Carmack, J. 2013. *Keynote at QUAKECON 2013,* https://www.youtube.com/watch?v=1PhArSujR_A

DARPA, 2013, DARPA CYBER GRAND CHALLENGE COMPETITOR PORTAL, https://cgc.darpa.mil

DARPA, 2015, DARPA SEEKS TO CREATE SOFTWARE SYSTEMS THAT COULD LAST 100 YEARS, http://www.darpa.mil/NewsEvents/Releases/2015/04/08.aspx

Ford, N. 2013. *Functional thinking: Why functional programming is on the rise* . http://www.ibm.com/developerworks/library/j-ft20/

Gat 2000. Erann Gat, Point of view: *Lisp as an alternative to Java*, Intelligence, Volume 11 Issue 4, Dec. 2000 Pages 21-24

Hänisch et al. 2014 *"Using a Sensor Network for Energy Optimization of Paper Machine Dryer Sections"* Athens Journal of Technology Engineering, Vol. 1, No. 3, September 2014

Harrison, W. 2000 *N=1, an Alternative for Software Engineering Research?* Proc. Workshop Beg, Borrow, or Steal: Using Multidisciplinary Approaches in Empirical Software Eng. Research, Int'l Conf. Software Eng., Aug. 2000.

Khare, S. et al, 2015. *Functional Reactive Stream Processing for Data-centric Publish/Subscribe Systems*, https://community.rti.com/paper/functional-reactive-stream-processing-data-centric-publishsubscribe-systems

Namiot, D, Sneps-Sneppe, M. 2014. *On IoT Programming,* International Journal of Open Information Technologies ISSN: 2307-8162 vol. 2, no. 10, 2014

Newton, R., Welsh, M. 2004, Region streams: functional macroprogramming for sensor networks, Proceeedings of the 1st international workshop on Data management for sensor networks: in conjunction with VLDB 2004 Pages 78-87

Ray, B. et al. 2014. "A Large Scale Study of Programming Languages and Code Quality in Github" Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2014

Reese. 2015. *A Comparison of Open Source Hardware: Intel Galileo vs. Raspberry Pi*. Technical Report. Mouser Electronics, http://www.mouser.de/applications/open-source-hardware-galileo-pi/.

Schneier, B. 2010. *The Dangers of a Software Monoculture* . https://www.schneier.com/essays/archives/2010/11/the_dangers_of_a_sof.html

Subramaniam, V. 2014, *Functional Programming in Java: Harnessing the Power of Java 8 Lambda Expressions*, O'Reilly.

Watson, A. H. and McCabe, T. J. 1996. *Structured testing: A testing methodology using the cyclomatic complexity metric.* NIST Special Publication, 1996.